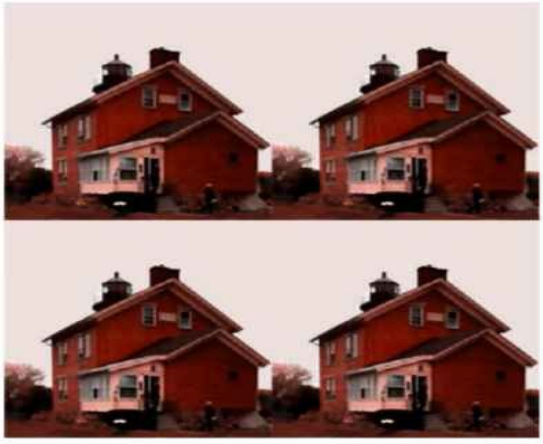
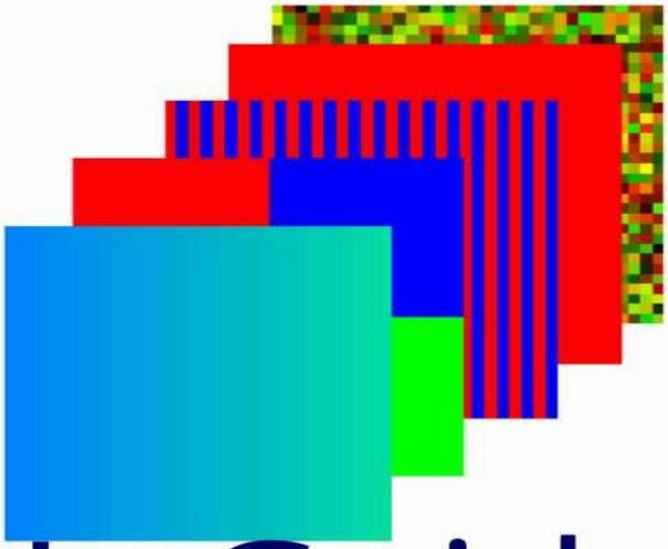
# WebGL Textures & Vertices





## Beginner's Guide

A. Butler

```
<!doctype HTML>

<html>

<head>

<meta http-equiv="content-type" content="text/html; charset=utf-8" />

<title>WebGL Texture</title>

<meta
    name="description"
    content="WebGL Texture."
/>

<script type="text/javascript"
    src="../GLEntity.js"
>
</script>

<script type="text/javascript"
```

```
    src="../GLControl.js"
>
</script>

<script type="text/javascript"
    src="GLSquare.js"
>
</script>

</head>

<body onload="new GLSquare
                                        (
'assets/lighthouse.jpg'
)"
>

<div>
<div id="eDebug">
</div>
```

```
<button id="animStop">
    Stop
</button>

<button id="animStart">
    Rotate
</button>

<canvas id="cv"
    width="512"
    height="512"
>
    Your browser doesn't support canvas.
</canvas>
</div>

</body>
</html>
var aVertices = new Float32Array(
                                                [
```

```
// left top:
// X,Y,Z:
-1.0, 1.0, 0.0,
// S,T:
0.0,1.0,

// right top:
// X,Y,Z:
1.0, 1.0, 0.0,
// S,T:
1.0, 1.0,

// right bottom:
// X,Y,Z;
1.0, -1.0, 0.0,
// S,T:
1.0, 0.0,

// left bottom
// X,Y,Z:
-1.0, -1.0, 0.0,
```

```
// S,T:
0.0, 0.0,
                                                    ]
                                                    );
var aIndices = new Uint16Array([
    3,2,0
                                                    ]);
var aIndices = new Uint16Array([
// triangle 1
```

**3,2,0,**

// triangle 2

# 0,2,1,

```
                                        ]);
var GLSquare = function(s)
                                        {
//Implementation...
                                        }
<body onload="new GLSquare
                                        (
'assets/lighthouse.jpg'
)"
>
var aIm = new Array();


var n = new GLEntity
                                        (
s,
```

);

aIm.push(n);

var controller = new GLControl

(

aVertices,

aIndices,

aIm,

this

);

nXLeft = Number(0.1);

nYTop = Number(0.9);

nXRight = Number(0.6);

nYBottom = Number(0.4);

var aVertices = new Float32Array(

[

// left top front

// index 0.

// X,Y,Z:

-1.0, 1.0, 0.0,

```
// S,T:
nXLeft,nYTop,
// right top front
// index 1.
// X,Y,Z:
1.0, 1.0, 0.0,
// S,T:
nXRight, nYTop,
// right bottom front
// index 2.
// X,Y,Z;
1.0, -1.0, 0.0,
// S,T:
nXRight, nYBottom,
// left bottom front
// index 3.
// X,Y,Z:
-1.0, -1.0, 0.0,
// S,T:
nXLeft, nYBottom,
]
```

```
                                                    );
var aVertices = new Float32Array(
                                                    [
// Top left quadrant:
-1.0, 1.0, 0.0,
0.0,1.0,

0.0, 1.0, 0.0,
1.0, 1.0,

0.0, 0.0, 0.0,
1.0, 0.0,

-1.0, 0.0, 0.0,
0.0, 0.0,

//Top right quadrant:
0.0, 1.0, 0.0,
0.0,1.0,

1.0, 1.0, 0.0,
```

1.0, 1.0,

1.0, 0.0, 0.0,

1.0, 0.0,

0.0, 0.0, 0.0,

0.0, 0.0,

//Bottom right quadrant:

0.0, 0.0, 0.0,

0.0,1.0,

1.0, 0.0, 0.0,

1.0, 1.0,

1.0, -1.0, 0.0,

1.0, 0.0,

0.0, -1.0, 0.0,

0.0, 0.0,

```
//Bottom left quadrant:
-1.0, 0.0, 0.0,
0.0,1.0,

0.0, 0.0, 0.0,
1.0, 1.0,

0.0, -1.0, 0.0,
1.0, 0.0,

-1.0, -1.0, 0.0,
0.0, 0.0,
                                                    ]
                                                    );
var aIndices = new Uint16Array([
    // upper left quadrant
3,2,0,
```

# 0,2,1,

// upper right quadrant

7,6,4,

**4,6,5,**

// lower right quadrant

11,10,8,

**8,10,9,**

// lower left quadrant

15,14,12,

**12,14,13,**

]);

var aVertices = new Float32Array(

[

// left top

-1.0, 1.0, 0.0,

0.0,4.0,

// right top

1.0, 1.0, 0.0,

4.0, 4.0,

// right bottom

1.0, -1.0, 0.0,

4.0, 0.0,

// left bottom

-1.0, -1.0, 0.0,

0.0, 0.0,

]);

```
var aVertices = new Float32Array(
                                        [
// left top
-1.0, 1.0, 0.0,
0.0,8.0,

// right top

1.0, 1.0, 0.0,
2.0, 8.0,

// right bottom
1.0, -1.0, 0.0,
2.0, 0.0,

// left bottom
-1.0, -1.0, 0.0,
0.0, 0.0,
                                        ]);
var aIndices = newUint16Array
                                        ([
```

3,2,0,

**0,2,1,**

]);

gl.texParameteri

(

gl.TEXTURE_2D,

gl.TEXTURE_WRAP_S,

gl.REPEAT

);

gl.texParameteri

(

gl.TEXTURE_2D,

gl.TEXTURE_WRAP_S,

gl.REPEAT

);

var n = new GLEntity(null,0);

n.img = this.generateGradientBG();

var controller = ev.currentTarget.controller;

// Reference to

// GLTexProcedure:

var glDemo = controller.glDemo;

```
var entity = controller.aEntities[0];
for (i = 0; i < this.nBuffer; i+=3){
u8a[i] = 255; // Red
u8a[i + 1] = 0; // Green
u8a[i + 2] = 0; // Blue
                                                    }
for (var i = 0; i < this.nBuffer; i = i+3){
// Each row r,g,b{0..95}
// Each row one pixel {0..31}
j = i % 96;
k = 95 - j;

// Assign Red, Green,
// Blue values:
u8a[i] = 0;
u8a[i + 1] = 255-k;
u8a[i + 2] = 255-j;
                                                    }
for (var i = 0; i < this.nBuffer; i = i+3){
var j = i % 2;
// Red stripe:
```

```
if (j == 0){
    u8a[i] = 255; // Red
    u8a[i+1] = 0;
    u8a[i+2] = 0;
                                                }
// Blue stripe:
else {
    u8a[i] = 0;
    u8a[i+1] = 0;
    u8a[i+2] = 255; // Blue.
                                                }
                                                }
for (var i = 0; i < this.nBuffer; i = i+3){
// Default black:
nRed = Number(0);
nGreen = Number(0);
nBlue = Number(0);

// pixels display
// in the top half if
// less than
```

```
// half the size
// of the entire buffer.
bTop = i < nHalf;

// j equals
// column number:
var j = i % 96;
bLeft = j < 48;

// Top tiles.
if (bTop){

    // Top left tile:
    if (bLeft){
    nRed = Number(255);
                                        }
    // Top right tile.
    else {
    nBlue = Number(255);
                                        }
                                        }
```

```
    // Bottom tiles.
    else {

    // Bottom left.
    // Blue+Red=Violet.
    if (bLeft){
    nRed = Number(255);
    nBlue = Number(255);
                                        }
    // Bottom right tile.
    else {
    nGreen = Number(255);
                                        }
                                        }
// Assign three
// color channels
// for one pixel.
u8a[i] = nRed;
u8a[i+1] = nGreen;
u8a[i+2] = nBlue;
```

```
                                                    }
for (var i = 0; i < this.nBuffer; i = i+3){
// Red:
u8a[i] = Math.random()*255;
// Green:
u8a[i + 1] = Math.random()*255;
// Blue:
u8a[i + 2] = 0;
                                                    }
gl.texImage2D(
gl.TEXTURE_2D,
```

**0,**

gl.RGB,

32,

32,

```
0,
gl.RGB,
gl.UNSIGNED_BYTE,
entity.img
                                                );
gl.activeTexture
                                                (
gl.TEXTURE0 + entity.idx
                                                );
gl.uniform1i
                                                (
entity.uSampler,
entity.idx
                                                );
getGLContext: function(canvas){

// Windows Phone 8.1
// default browser
// uses 'experimental-webgl'.
var a3D = ['webgl',
```

```
'experimental-webgl',
'webkit-3d',
'moz-webgl'
                                                    ];

var glContext = null;

try {

// Iterate over our array.
for (var i = 0; i < a3D.length; i++) {

// Try to obtain a 3D context.
glContext = canvas.getContext(a3D[i]);

// If we found a context,
// then break out of the loop.
    if (glContext != null) {
    break;
                                                    }
                                                    }
```

```
                                        }

// If there's an error,
// then display it.
catch(err) {
this.viewError(err,this);
                                        }

// WebGLContext or null.
return glContext;
},
var shaderF = this.getShader
                                        (
"shader-f",
gl.FRAGMENT_SHADER
                                        );
var shaderV = this.getShader
                                        (
"shader-v",
gl.VERTEX_SHADER
                                        );
```

```
var p = gl.createProgram();
gl.attachShader
                                        (
p,
shaderF
                                        );
gl.attachShader
                                        (
p,
shaderV
                                        );
gl.linkProgram
                                        (
p
                                        );
gl.useProgram
                                        (
p
                                        );
shader = gl.createShader
                                        (
```

```
nType
                                                    );
sCode = "attribute vec4 a_position;"
+"attribute vec2 a_tex_coord0;"
+"varying vec2 v_tex_coord0;"

+"uniform mat4 um4_matrix;"
+"uniform mat4 um4_pmatrix;"

+"void main(void) {"
+ "gl_Position = um4_pmatrix um4_matrix a_position;"
+ "v_tex_coord0 = a_tex_coord0;"
                                                    +"}";
sCode = "precision mediump float;"
+"uniform sampler2D u_sampler0;"
+"varying vec2 v_tex_coord0;"

+" void main(void) {"
+"gl_FragColor = texture2D(u_sampler0, v_tex_coord0);"
                                                    +"}";
gl.shaderSource
```

(

shader,

sCode

);

gl.compileShader

(

shader

);

if (!gl.getShaderParameter

(

shader,

gl.COMPILE_STATUS

))

{ ... }

var sError = gl.getShaderInfoLog

(

shader

);

1. attribute vec4 a_position;

2. attribute vec2 a_tex_coord0;

3. varying vec2 v_tex_coord0;

4. uniform mat4 um4_matrix;

5. uniform mat4 um4_pmatrix;

6. void main(void) {

7. gl_Position = um4_pmatrix *um4_matrix* a_position;

8. v_tex_coord0 = a_tex_coord0;

9.}

1. precision mediump float;

2. uniform sampler2D u_sampler0;

3. varying vec2 v_tex_coord0;

4. void main(void) {

5. gl_FragColor = texture2D(u_sampler0, v_tex_coord0);

6. }

matrixRotationY: function (m,y){

var c = Math.cos(y);

var s = Math.sin(y);

return [

c, m[1], s, m[3],

```
m[4], m[5], m[6], m[7],
-s, m[9], c, m[11],
m[12], m[13], m[14], m[15],
                                        ];
},
matrix = controller.matrixRotationY
                                        (
matrix,
controller.nRad
                                        );
...
controller.nRad += controller.N_RAD;
gl_Position = um4_pmatrix um4_matrix a_position;
var uMP = gl.getUniformLocation
                                        (
program,
"um4_pmatrix"
                                        );
var aMP = new Float32Array(
                                        [
   2.4,0,0,0,
```

```
    0,2.4,0,0,
    0,0,-1,-1,
    0,0,-0.2,0]
                                        );
gl.uniformMatrix4fv
                                        (
uMP,
gl.FALSE,
aMP
                                        );
this.uMatrixTransform = gl.getUniformLocation
                                        (
    program,
    "um4_matrix"
                                        );
this.aPosition = gl.getAttribLocation
                                        (
    program,
    "a_position"
                                        );
gl.enableVertexAttribArray
```

(

this.aPosition

);

gl.viewport

(

0,

0,

512,

```
                                        );
this.nBufferLength = Number
                                        (
aI.length
                                        );
gl.bindBuffer
                                        (
gl.ARRAY_BUFFER,
WebGLBuffer
                                        );
gl.bufferData
                                        (
gl.ARRAY_BUFFER,
aV,
gl.STATIC_DRAW
                                        );
vertexAttribPointer
                                        (
Number index,
```

```
Number size,
Number type,
boolean normalized,
Number stride,
Number offset
)
gl.vertexAttribPointer
                                                    (
this.aPosition,
```

**3,**

gl.FLOAT,

gl.FALSE,

20,

```
                                                    );
gl.bindBuffer
                                                    (
gl.ELEMENT_ARRAY_BUFFER,
bIndices
                                                    );
gl.bufferData
                                                    (
gl.ELEMENT_ARRAY_BUFFER,
aI,
gl.STATIC_DRAW
                                                    );
for (var i = 0; i < aEntities.length; i++){
// GLEntity at 'i'.
var entity = aEntities[i];

// Increment property nImagesToLoad:
this.nImagesToLoad += entity.getImageVariables
                                                    (
```

```
this
                                                    );
                                                    }
this.uSampler = gl.getUniformLocation
                                                    (
program,
"u_sampler"+this.idx
                                                    );
this.aTexCoord = gl.getAttribLocation
                                                    (
program,
"a_tex_coord"+this.idx
                                                    );
gl.vertexAttribPointer
                                                    (
this.aTexCoord,
```

# 2,

gl.FLOAT,

gl.FALSE,

20,

```
                                        );
gl.enableVertexAttribArray
                                        (
this.aTexCoord
                                        );
this.img.controller = controller;
this.img.entity = this;
this.img.onload = this.setImage;
this.img.src = this.sSrc;
if (ev.currentTarget != null){
controller = ev.currentTarget.controller;
entity = ev.currentTarget.entity;
                                        }
gl.uniform1i
                                        (
entity.uSampler,
entity.idx
                                        );
gl.activeTexture
```

```
                                        (
gl.TEXTURE0 + entity.idx
                                        );
gl.bindTexture
                                        (
    gl.TEXTURE_2D,
    entity.texture
                                        );
gl.pixelStorei
                                        (
gl.UNPACK_FLIP_Y_WEBGL,
true
                                        );
gl.texImage2D(
gl.TEXTURE_2D,
```

**0,**

gl.RGBA,

gl.RGBA,

gl.UNSIGNED_BYTE,

entity.img

);

gl.texParameteri

(

gl.TEXTURE_2D,

gl.TEXTURE_WRAP_S,

gl.CLAMP_TO_EDGE

);

gl.texParameteri

(

gl.TEXTURE_2D,

gl.TEXTURE_WRAP_T,

gl.CLAMP_TO_EDGE

);

gl.texParameteri

(

```
gl.TEXTURE_2D,
gl.TEXTURE_MIN_FILTER,
gl.NEAREST
                                        );
gl.texParameteri
                                        (
gl.TEXTURE_2D,
gl.TEXTURE_MAG_FILTER,
gl.NEAREST
                                        );
if (controller.nImagesToLoad <= controller.nImagesLoaded){
   controller.setProgramVariables(controller);
                                        }
gl.validateProgram
                                        (
program
                                        );
if (!gl.getProgramParameter
                                        (
   program,
   gl.VALIDATE_STATUS
```

```
    ))
                                                {
// The program failed.
                                                }
var validateError = gl.getProgramInfoLog
                                                (
program
                                                );
if (glDemo.init != null){
glDemo.init(controller);
                                                }
gl.uniformMatrix4fv
                                                (
controller.uMatrixTransform,
gl.FALSE,
new Float32Array(matrix)
                                                );
gl.drawElements
                                                (
gl.TRIANGLES,
controller.nBufferLength,
```

gl.UNSIGNED_SHORT,

);

<body onload="new GLSquare

(

'assets/fish.jpg'

)"

>

cv.addEventListener

(

'click',

this.animOne,

false

);

var animStart = document.getElementById

(

"animStart"

);

var animStop = document.getElementById

(

"animStop"

```
                                                    );
animStart.controller = this;
animStop.controller = this;
animStart.addEventListener
                                                    (
'click',
this.animStart,
false
                                                    );

animStop.addEventListener
                                                    (
'click',
this.animStop,
false
                                                    );
window.requestAnimationFrame = window.requestAnimationFrame
|| window.mozRequestAnimationFrame
|| window.webkitRequestAnimationFrame
|| window.msRequestAnimationFrame;
animStart: function(ev){
```

```
// Assume 'ev' type
// is GLControl.
var controller = ev;

// If ev.currentTarget is
// is valid, then ev is a
// MouseEvent.
if (ev.currentTarget != null){
    controller = ev.currentTarget.controller;
                                                }
...
controller.frameAnimID = window.requestAnimationFrame(
function() {
    controller.drawScene(
    controller
                                                );
                                                 }
                                                );
// Call the demo project's
// renderer for display.
if (glDemo.render != null){
```

```
this.render = glDemo.render;
                                        }

// Use the default render
// method to draw the scene.
else {
this.render = this.renderDefault;
                                        }
matrix = controller.matrixRotationY
                                        (
   matrix,
   controller.nRad
                                        );
gl.uniformMatrix4fv
                                        (
controller.uMatrixTransform,
gl.FALSE,
new Float32Array(matrix)
                                        );
gl.drawElements
                                        (
```

```
gl.TRIANGLES,
controller.nBufferLength,
gl.UNSIGNED_SHORT,
```

```
                                                    );
/* "use strict" enforces
    aspects of good syntax.
    Changes some silent
    JavaScript errors
    to exceptions.
    Corrects some JavaScript
    errors, thereby optimizing
    the code.
    Prevents use of keywords
    reserved for future
    revisions to the JavaScript
    specification.
                                                    */
"use strict";

                                                    /**
* JavaScript Controller 'class'
* includes methods
```

* and variables

* shared by examples provided

* with the e-book series

* "Online 3D Media with WebGL".

* @param aVert:

* Float32Array of

* vertex coordinates.

* interleaved with texels.

* @param aIdx:

* Uint16Array of indices into

* the vertex array.

* @param glDemo:

* Maintain a reference

* to an Object prepared

* for each unique example

* provided with the e-book

* series.

* Composition of classes:

```
* GLControl.glDemo.


* @param aE: Array if
* GLEntity Objects
* defined for the
* e-book series.


* @returns: GLControl instance.
                                                    */
var GLControl = function(aVert,aIdx,aE,glDemo){
// Amount to increment
// rotation in radians
// per animated frame.
this.N_RAD = new Number(0.5);


// The maximum number of
// frames to animate.
this.FRAME_MAX = Number(512);


// The current display
// frame while an animation
```

```
// runs.
this.frameCount = Number(0);

// The starting angle
// of rotation in radians.
// Increments per animation
// frame.
this.nRad = new Number(0);

// Array of GLEntity.
this.aEntities = aE;

// The current demo
// or WebGL example
// 'class' Object.
this.glDemo = glDemo;

// Call the demo project's
// renderer for display.
if (glDemo.render != null){
    this.render = glDemo.render;
```

```
                                                    }

// Use the default render
// method to draw the scene.
else {
    this.render = this.renderDefault;
                                                    }

// JavaScript property
// reference to the
// vertex shader's
// uniform mat4
// used for animated
// matrix transformations.
this.uMatrixTransform;

// One frame's identification
// number provided by
// JavaScript API
// requestAnimationFrame().
this.frameAnimID = Number(0);
```

```
// Animation timing properties
// follow.

// Time between frames.
this.FRAME_INTERVAL = Number(128);

// Current frame time.
this.frameCurrent = Number(0);
// Previous frame time.
this.framePrevious = Number(0);

// requestAnimationFrame should be
// implemented within all browsers
// which support WebGL.
// However currently
// different browsers, use
// different method names
// for similar
// functionality.
window.requestAnimationFrame = window.requestAnimationFrame ||
```

```
window.mozRequestAnimationFrame ||
window.webkitRequestAnimationFrame ||
window.msRequestAnimationFrame;

// Save a reference to
// HTML element for
// display of debugging output.
this.eDebug = document.getElementById("eDebug");

// Obtain a reference
// to the current page's
// canvas.
var cv = document.getElementById('cv');

// Save the WebGLContext.
this.gl = this.getGLContext(cv);
if (this.gl == null){
    // Show errors.
    this.viewError(
    "Error initializing WebGL.",
    this
```

```
                                                    );
    return null;
                                                    }

// Compile two WebGL shaders
// then link to one WebGLProgram.
this.program = this.getProgram();

// If the program's null
// then tell the user,
// and return.
if (this.program == null){
    this.viewError(
    "Error initializing WebGL.",
    this
                                                    );
    return null;
                                                    }

// Save shader
// variable and
```

```
// uniform locations.
// Upload perspective
// matrix.
this.getProgramVariables();

// Generate WebGLBuffers.
// Assign values, upload
// buffer values to the shader.
this.getBuffers(aVert,aIdx);

// Download Image files
// when necessary.
// Obtain per image
// shader properties.
this.getImages();

// Assign event listeners
// to the canvas, window,
// start and stop buttons.
this.setListeners(cv);
```

```
// Return our new controller.
return this;
} // End GLControl constructor.
                                        /**
* Prototype
* methods to
* initialize WebGL
* properties.
                                         */
GLControl.prototype = {

                                        /**
* Compile and link
* a fragment and shader
* WebGLShader
* to a WebGLProgram.
                                         *
* @returns WebGLProgram
                                         */
getProgram: function(){
```

```
// Save the WebGLContext
var gl = this.gl;

// Obtain and compile
// the fragment shader.
var shaderF = this.getShader
                                        (
    "shader-f",
    gl.FRAGMENT_SHADER
                                        );

// Obtain and compile
// the vertex shader.
var shaderV = this.getShader
                                        (
    "shader-v",
    gl.VERTEX_SHADER
                                        );

// Create an empty
// WebGLProgram Object.
```

```
var p = gl.createProgram();

// Attach our
// fragment shader
// to the program.
gl.attachShader
                                        (
    p,
    shaderF
                                        );

// Attach our
// vertex shader
// to the program.
gl.attachShader
                                        (
    p,
    shaderV
                                        );

// Link the
```

```
// shaders to
// the program.
gl.linkProgram
                                        (
    p
                                        );

// Assign the
// program for
// use. Future
// WebGL draw
// operations
// act upon this
// program.
gl.useProgram
                                        (
    p
                                        );

// Return program.
// Save the reference.
```

```
// with this controller.
return p;
},
                                                    /**
* Generate WebGLBuffer Objects.
                                                     *
* @param aV: Float32Array
* representing vertex X,Y,Z
* coordinates interleaved
* with texel ST coordinates.

* @param aI:
* Uint16Array of indices
* for element array
* buffer.

* @ returns nothing.
                                                    */
getBuffers: function(aV,aI){

//Local WebGLContext.
```

```
var gl = this.gl;

// Generate a WebGLBuffer object.
var bufferVT = gl.createBuffer();

// Tell WebGL we
// want to use bufferVT
// as an array buffer.
// The buffer
// contains data
// for direct or
// indirect access
// within the shaders.
gl.bindBuffer
                                        (
   gl.ARRAY_BUFFER,
   bufferVT
                                        );

// Upload array of
// vertex and texel
```

```
// coordinates to
// the GPU once.
// STATIC_DRAW
// means we plan
// to re use the
// data as is,
// without dynamic
// modifications.
gl.bufferData
                                        (
    gl.ARRAY_BUFFER,
    aV,
    gl.STATIC_DRAW
                                        );

// Tell the GPU which
// parts of the buffer
// to assign to which
// attribute.
// Here we layout
// vertex attributes.
```

```
// GLEntity.js will
// layout texel attributes.

// 1. Attribute location
// to receive buffer data.

// 2. Three array
// elements to assign
// to this particular attribute.
// Three coordinates
// per vertex.

// 3. gl.FLOAT:
// Type of information
// for each array element
// is float.

// 4. false: Values in the buffer
// don't need to be
// normalized.
```

```
// 5. Stride between
// attributes within the
// buffer.
// How many bytes
// between start of the first
// attribute and the start
// of the second attribute.
gl.vertexAttribPointer
                                        (
    this.aPosition,
    3,
    gl.FLOAT,
    gl.FALSE,
    20,
    0
                                        );

    // Save the length
    // of the element buffer
    // for drawing with
    // drawElements() later.
```

```
this.nBufferLength = Number(aI.length);

// Generate another
// WebGLBuffer Object.
var bIndices = gl.createBuffer();

// This time tell
// the GPU we want
// the buffer
// to operate
// as an element array.
gl.bindBuffer
                                        (
gl.ELEMENT_ARRAY_BUFFER,
bIndices
                                        );

// Upload the data
// from our index
// array to the GPU.
gl.bufferData
```

```
                                                (
    gl.ELEMENT_ARRAY_BUFFER,
    aI,
    gl.STATIC_DRAW
                                                );

},
                                                /**
* Process shader variables.
* Obtain references to variables
* from the shaders.
* Upload a perspective projection
* matrix.
* @return nothing.
                                                */
getProgramVariables: function(){

// Local WebGLContext.
var gl = this.gl;

// Local WebGLProgram
```

```
var program = this.program;

// We only assign the
// perspective projection
// matrix once.
// First get the location
// within our vertex shader
// of the 4x4 matrix to
// apply perspective projection.
var uMP = gl.getUniformLocation
                                        (
    program,
    "um4_pmatrix"
                                        );

// Create a
// Perspective matrix
// with Float32Array type.
// In effect the matrix
// will modify vertex
// locations to appear
```

```
// in perspective.
// In other words, more
// distant vertices appear
// closer to a vanishing point.
var aMP = new Float32Array(
                                          [
    2.4,0,0,0,
    0,2.4,0,0,
    0,0,-1,-1,
    0,0,-0.2,0]
                                          );

// Upload the
// JavaScript
// perspective matrix 'aMP'
// to shader uniform location 'uMP'.
// The middle parameter
// must be set to 'false'
// Meaning don't transpose
// the matrix when uploaded.
gl.uniformMatrix4fv
```

(

uMP,

gl.FALSE,

aMP

);

// Obtain and save the

// location of the matrix

// within our vertex shader.

// During draw operations

// modify then upload

// uMatrixTransform, to

// rotate the mesh or meshes.

this.uMatrixTransform = gl.getUniformLocation

(

program,

"um4_matrix"

);

// The location of the

// attribute in our

```
// vertex shader named
// a_position,
// saved to JavaScript
// aPosition property.
this.aPosition = gl.getAttribLocation
                                        (
    program,
    "a_position"
                                        );

// Tell WebGL to
// activate aPosition.
// aPosition references
// the shader attribute
// 'a_position'.
// Now a_position is
// set to receive a
// stream of vertex
// data from our
// vertex buffer.
gl.enableVertexAttribArray
```

(

this.aPosition

);

// Tell WebGL the size

// of our view port once.

// The book's examples

// always use a 512 by 512

// view port.

gl.viewport

(

0,

0,

512,

```
                                        );
},

                                        /**
* Prepare to
* download and process
* images for textures.
                                        */
getImages: function() {

    // Local variable to
    // the controller's array
    // of GLEntity.
    var aEntities = this.aEntities;

    // Local variable reference
    // to the WebGLContext
    var gl = this.gl;
```

```
// Keep track of how many
// images we need to download.
this.nImagesToLoad = Number(0);

// Track how many
// images have downloaded.
// Processing completes
// after all the images
// download.
this.nImagesLoaded = Number(0);

// Iterate over our array of GLEntity.
for (var i = 0; i < aEntities.length; i++){
var tex = aEntities[i];

// Process data for each GLEntity.
// Some images already have image data.
// No need to download the image file.
// Some image files require download.
// Therefore getImageVariables()
// returns 1 if the image must download
```

```
    // and 0 otherwise.
    this.nImagesToLoad += tex.getImageVariables(this);
                                                }

    // We don't need to download
    // computer generated textures,
    // just set parameters
    // for display.
    if (this.nImagesToLoad == 0){

    // Here simply call
    // GLEntity.setImage()
    // directly.
    this.aEntities[0].setImage(this);
                                                }
},
                                                /**
* Activates after all
* images complete
* asynchronous
* downloads.
```

```
* Some browsers
* require validation
* after setting
* active textures.
                                                *
* @param controller: Reference
* to GLControl object.
                                                */
setProgramVariables: function(controller){
    var gl = controller.gl;
    var program = controller.program;
    var glDemo = controller.glDemo;

    // WebGL verify
    // accuracy of shaders.
    gl.validateProgram
                                                (
    program
                                                );
```

```
// WebGL API method
// getProgramParameter().
// First argument is the
// program we want to check.
// Second argument is what
// we want to check.
// We want the status of
// the preceding validation.
// Returns false if
// the program's invalid.
if (!gl.getProgramParameter
                                        (
program,
gl.VALIDATE_STATUS
)
)
                                        {

// Perhaps one of the most useful
// calls. Find out what went
// wrong in the shaders.
```

```
// The WebGL API method
// getProgramInfoLog().
// Returns a String
// with information.
var validateError = gl.getProgramInfoLog
                                        (
program
                                        );

// See the error
// in the Web page.
controller.viewError
                                        (
"Error while compiling the program:" + validateError,
controller
                                        );

// WebGL API method
// deleteProgram(),
// detaches any attached
// shaders. Assigns the
```

```
    // program for deletion.
    gl.deleteProgram(program);
    return;
                                                }

    // Some projects
    // include one time
    // initialization
    // method.
    // Must happen
    // after textures
    // initialize.
    if (glDemo.init != null){
    glDemo.init(controller);
                                                }

    // If everything validated
    // now display one frame
    // of the animation.
    controller.animOne(controller);
},
```

```
                                        /**
* Show one frame
* of the animation.
* Assign the frameCount
* the maximum number of
* frames per animation,
* minus one.
                                         *
* @param ev: Either MouseEvent
* or GLControl reference.
                                         */
animOne: function(ev){
    // Assume 'ev' is
    // of type GLControl.
    var controller = ev;

    // If ev has a 'currentTarget'
    // property, then
    // currentTarget.controller
    // is a reference to
    // GLControl.
```

```
    if (ev.currentTarget != null){
    controller = ev.currentTarget.controller;
                                                }

    // Animations run from
    // 0 to FRAME_MAX.
    // Specify we only
    // want to see one frame.
    controller.frameCount = controller.FRAME_MAX - 1;

    // requestAnimationFrame() has
    // one parameter which is the call back.
    // We'll call drawScene().
    controller.frameAnimID = window.requestAnimationFrame(
    function() {
    controller.drawScene(controller);
                                                }
                                                );
},
                                                /**
* Start the animation.
```

```
* Call vendor (browser)
* specific implementation
* of requestAnimationFrame().
* Process a frame, then
* request another frame.
                                                    *
* @param ev: Either a reference
* to GLControl, or
* an MouseEvent object.
* Depends on who calls
* animStart().
                                                    */
animStart: function(ev){

// Assume 'ev' type
// is GLControl.
var controller = ev;

// If ev.currentTarget is
// valid, then ev is of
// type MouseEvent, and
```

```
// currentTarget.controller
// is a GLControl
// reference.
if (ev.currentTarget != null){
    controller = ev.currentTarget.controller;
                                                }
controller.eDebug.innerHTML = "";

// Don't start multiple
// animations.
// The frameAnimID is only
// zero, when the animation's
// not running.
if (controller.frameAnimID == 0){

    // The current frame
    // count.
    // The animation frame
    // range is
    // {0...FRAME_MAX}.
    controller.frameCount = Number(0);
```

```
    // The current frame
    // starts now.
    // framePrevious used
    // for timing between
    // frames.
    controller.frameCurrent = controller.framePrevious = Date.now();

    // Return the ID of the
    // current frame.
    // requestAnimationFrame()
    // triggers
    // drawScene().
    controller.frameAnimID = window.requestAnimationFrame(
    function() {
    controller.drawScene(controller);
                                                    }
                                                    );
                                                    }
},
                                                    /**
```

```
* Stop the animation.
* param ev: Either a MouseEvent
* Object or a GLControl reference.
                                                    */
animStop: function(ev){
var controller = ev;

if (ev.currentTarget != null){
    controller = ev.currentTarget.controller;
                                                    }

// Signal we've drawn
// all animation frames.
controller.frameCount = controller.FRAME_MAX;

// frameAnimID equals zero
// when the animation isn't
// running.
controller.frameAnimID = Number(0);
},
                                                    /**
```

```
* Obtain a reference to
* the WebGL context.
* @param canvas: HTML5 canvas element.
* @returns: Either a valid
* WebGLContext or null.
                                                */
getGLContext: function(canvas){

// Iterate over an array
// of potential names
// for the WebGLContext.

// Unfortunately not all
// browsers use 'webgl'.
// For example Windows Phone 8.1
// default browser uses
// 'experimental-webgl'.
var a3D = ['webgl',
    'experimental-webgl',
    'webkit-3d',
    'moz-webgl'
```

```
                                                    ];
var glContext = null;

try {

    // Loop over our array.
    for (var i = 0; i < a3D.length; i++) {

    // Try to obtain a 3D context.
    glContext = canvas.getContext(a3D[i]);

    // If we found a context,
    // then break out of the loop.
    if (glContext != null) {
    break;
                                                    }
                                                    }
                                                    }

// If there's an error,
// then display it.
```

```
catch(err) {
this.viewError(err,this);
                                        }

// WebGLContext or null.
return glContext;
},
                                        /**
* Returns one compiled
* WebGL shader Object.

* @param sID: String id of
* the HTML element with
* shader code.
                                        *
* @param nType: Either
* gl.FRAGMENT_SHADER, or
* gl.VERTEX_SHADER.
                                        *
* @returns: a WebGLShader
* or null if compilation failed.
```

```
                                                */
getShader: function(sID, nType) {
var gl = this.gl;

// String of
// shader code.
var sCode = "";

// Shader element
// from Web page.
var eShader = null;

var nodeText;

// WebGLShader
var shader = null;

// Get element
// containing
// shader code.
eShader = document.getElementById(sID);
```

```
// If the shader is not
// declared within the Web
// page, then use the default.
if (eShader == null) {
    if (nType == gl.FRAGMENT_SHADER){

    // Line 2: uniform sampler2D u_sampler0;
    // A sampler2D references
    // an active texture.

    // Line 3: varying vec2 v_tex_coord0;
    // Processes texels.

    // Line 4: void main(void)
    // Fragment shader's
    // entry point.

    // Built in function texture2D()
    // returns a sample from
    // a texture unit.
```

```
sCode = "precision mediump float;"
+"uniform sampler2D u_sampler0;"
+"varying vec2 v_tex_coord0;"

+" void main(void) {"
+"gl_FragColor = texture2D(u_sampler0, v_tex_coord0);"
                                                    +"}";

                                                      }

else {

// Default vertex shader.

// Line 1: attribute vec4 a_position
// receives X, Y, and Z coordinates
// from buffer of vertices and texels.

// Line 2: attribute vec2 a_tex_coord0
// receives the S and T texel coordinates
```

// from buffer of vertices and texels.

// Line 3: varying vec2 v_tex_coord0

// Passed to the GPU for interpolation,

// then on to the fragment shader.

// Line 4: uniform mat4 um4_matrix

// 4 x 4 matrix for

// transformations.

// Line 5: uniform mat4 um4_pmatrix

// 4 x 4 matrix for perspective

// projection.

// Line 7:

// gl_Position = um4_pmatrix *um4_matrix* a_position;

// gl_Position output for vertex

// location.

// Line 8: v_tex_coord0 = a_tex_coord0;

// passes the current texel

```
// to the GPU for interpolation.

sCode = "attribute vec4 a_position;"
+"attribute vec2 a_tex_coord0;"
+"varying vec2 v_tex_coord0;"

+"uniform mat4 um4_matrix;"
+"uniform mat4 um4_pmatrix;"

+"void main(void) {"
+ "gl_Position = um4_pmatrix um4_matrix a_position;"
+ "v_tex_coord0 = a_tex_coord0;"
                                        +"}";

                                          }

                                          }

// Retrieve shader from
// the current Web page.
else{
nodeText = eShader.firstChild;
```

```
// Some examples in the WebGL Texture
// series, declare vertex shader code
// within the current Web page.

// Iterate over
// Web page's code.
// Assign to
// local variable 'sCode'.
while(nodeText != null) {
if (nodeText.nodeType == nodeText.TEXT_NODE) {
sCode += nodeText.textContent;
                                        }

// Get the next row of characters.
nodeText = nodeText.nextSibling;
                                        }
                                        }

// Generate a WebGLShader of
// type FRAGMENT_SHADER or
```

```
// type VERTEX_SHADER.
shader = gl.createShader(nType);

// Assign the source
// String to the
// shader.
gl.shaderSource
                                        (
shader,
sCode
                                        );

// Compile the shader's
// source code.
gl.compileShader(shader);

// The WebGL API method
// getShaderParameter()
// verifies the
// shader compiled.
```

```
// getShaderParameter()
// returns false,
// if compile failed.
if (!gl.getShaderParameter
                                        (
shader,
gl.COMPILE_STATUS
))
                                        {

// The WebGL API getShaderInfoLog()
// Returns a String of
// information about
// the shader.
var sError = gl.getShaderInfoLog(shader);

// Display the
// error to the
// Web page.
this.viewError
                                        (
```

```
    "An error occurred compiling the shaders: " + sError,
    this
                                                );

    // Return null.
    // Signals calling
    // method there
    // was an error.
    return null;
                                                }

    // Return a
    // valid shader.
    return shader;
},
                                                /**
* Assign listeners to
* the canvas, window,
* animation start button,
* and animation stop button.
```

```
                                                    *
* @param cv: An HTML5 canvas element.
                                                    */
setListeners: function(cv){

// Create a 'controller'
// property and assign
// reference to
// this GLControl 'class'.
cv.controller = this;

// When the user
// taps the canvas,
// execute animOne().
cv.addEventListener
                                                    (
    'click',
    this.animOne,
    false
                                                    );
```

```
// Obtain references to
// the animStart and animStop
// buttons, from the Web page.
var animStart = document.getElementById
                                        (
    "animStart"
                                        );
var animStop = document.getElementById
                                        (
    "animStop"
                                        );

// Each button's controller
// property receives a
// reference to this GLControl.
animStart.controller = this;
animStop.controller = this;

// Execute animStart()
// when the user taps
// the animStart button.
```

```
animStart.addEventListener
                                        (
    'click',
    this.animStart,
    false
                                        );

// Execute animStop()
// when the user taps
// the animStop button.
animStop.addEventListener
                                        (
    'click',
    this.animStop,
    false
                                        );

// Create and assign
// controller property.
window.controller = this;
```

```
// When the window
// unloads, execute
// animStop().
window.addEventListener
                                        (
    "unload",
    this.animStop,
    false
                                        );
},
                                        /**
* Display error information
* to the Web page.
                                         *
* @param err: String with an error
* message.
* @param controller: Reference to
* GLControl.
                                        */
viewError: function (err,controller){
    controller.eDebug.innerHTML = "Your browser might not support WebGL.
```

```
<br />";

    controller.eDebug.innerHTML += "For more information see <a
href='http://get.webgl.org'>http://get.webgl.org</a>.<br />";

    controller.eDebug.innerHTML += err.toString();

},

                                                    /**

* Determine if enough

* time has passed

* between display frames

* of an animation.


* @param controller:

* Reference to GLControl.


* @return: Boolean true if it's time

* to display another frame.

* false if not enough time has

* passed between frames.

                                                    */

checkFrameTime: function(controller){


// Render only at
```

```
// Render only at
// specified intervals.
// Otherwise some
// devices
// draw too fast.
controller.frameCurrent = Date.now();

// Difference between
// the time the previous
// frame rendered,
// and now.
var delta = controller.frameCurrent - controller.framePrevious;

if (delta < controller.FRAME_INTERVAL){

    // Not enough time
    // has passed, return false.
    return false;
                                                }

// Save this frame time.
controller.framePrevious = controller.frameCurrent;
```

controller.framePrevious = controller.frameCurrent;

// Render a frame.

// Return true.

return true;

},

/**

* Default rendering method.

* Rotates the matrix from

* the first GLEntity

* in the array.

* Uploads the matrix.

* Then draws all

* elements in the

* element array buffer.

*

* @param: controller is a reference

* to GLControl.

* @returns nothing.

*/

renderDefault: function(controller){

```
var gl = controller.gl;


// Get the matrix.
var matrix = controller.aEntities[0].matrix;


// Rotate the
// matrix around
// the Y axis.
matrix = controller.matrixRotationY
                                        (
matrix,
controller.nRad
                                        );


// Upload the
// new matrix
// to the
// vertex shader.
gl.uniformMatrix4fv
                                        (
controller.uMatrixTransform,
```

```
    gl.FALSE,
    new Float32Array(matrix)
                                        );

    // Run the series
    // of vertices and texels
    // through attributes
    // in the vertex shader
    gl.drawElements
                                        (
    gl.TRIANGLES,
    controller.nBufferLength,
    gl.UNSIGNED_SHORT,
    0
                                        );

    // Increase the radians.
    // for rotation.
    controller.nRad += controller.N_RAD;
},
                                        /***
```

```
* Provides timing
* to render a scene.

* Calls the
* current projects
* render() method
* if one exists.
* otherwise calls
* the default render
* method.

* @param controller:
* Reference to GLControl.
                                                */
drawScene: function(controller) {

// Recursively call
// drawScene()
// if not enough
// time has passed.
if (controller.checkFrameTime(controller) == false){
```

```
    controller.frameAnimID = window.requestAnimationFrame(
    function()
                                                {
    controller.drawScene(controller);
                                                }
                                                );
    return;
                                                }

// Automatically stops
// animation after
// FRAME_MAX number of
// frames have rendered.
if (controller.frameCount < controller.FRAME_MAX){

    // Render one frame.
    controller.render(controller);

    // Increment the frame count.
    controller.frameCount++;
```

```
    // Recursion
    // process another frame.
    controller.frameAnimID = window.requestAnimationFrame(
    function()
                                                {
    controller.drawScene(controller);
                                                }
                                                );
                                                }
else {

    // Stop the animation
    // if FRAME_MAX
    // frames have rendered.
    controller.animStop(controller);
                                                }
},
                                                /**
* http://www.apache.org/licenses/LICENSE-2.0
* The following matrix transformation
* functions are licensed under the
```

```
* Creative Commons Attribution 3.0 License,
* and code samples are licensed under the Apache 2.0 License.
                                                    */
matrixRotationY: function (m,y){
var c = Math.cos(y);
var s = Math.sin(y);

return [
c, m[1], s, m[3],
m[4], m[5], m[6], m[7],
-s, m[9], c, m[11],
m[12], m[13], m[14], m[15],
                                                    ];
},
matrixRotationX: function (m,x){
var c = Math.cos(x);
var s = Math.sin(x);

return [
    m[0], m[1], m[2], m[3],
    m[4], c, -s, m[7],
```

```
    m[8], s, c, m[11],
    m[12], m[13],m[14],m[15],
                                        ];
}// The last function ends
// with no comma.

} // End GLControl prototype.
                                        /**
* GLEntity 'class' loads
* and prepares image data
* for display as WebGL textures.
                                         *
* GLEntity prepares shader
* attributes and uniforms
* to process a texture.
* Includes a matrix
* for transformation.

* Allows texture, matrix,
* and shader flexibility.
```

* Rendering methods

* may use one texture

* per GLEntity or

* share a texture

* among a list of GLEntity.

* May use one matrix

* per GLEntity, or share

* a matrix

* Shaders may use one

* attribute per GLEntity

* or share the attribute.

* Shaders may use one

* uniform per GLEntity

* or share the uniform.

* SHADER DEPENDENCIES:

* An array of GLEntity

* depend on a vertex shader

* with at least one attribute:

* named "a_tex_coord<number>"

```
* Where <number> equals the
* property : GLEntity.idx.
                                                    *
* An array of GLEntity
* depend on a fragment shader
* with at least one uniform:
* named "u_sampler<number>"
* Where <number> equals the
* property : GLEntity.idx.
                                                    *
* At least one GLEntity
* in an array of GLEntity,
* maintains a valid shader
* attribute and uniform.
                                                    */
var GLEntity = function(s,i){

// image file path
// source String.
this.sSrc = s;
```

```
// Image data.
// Either in array
// or Image Object
// form.
this.img = null;

// WebGLTexture
this.texture = null;

// Uniform Sampler
// for a WebGLTexture
this.uSampler = null;

// Attribute representing
// texture coordinates
// as S,T texels.
this.aTexCoord = null;

// The index for this
// particular GLEntity.
// Required to process
```

```
// unique uniforms,
// textures, and
// set the active texture.
this.idx = parseInt(i);

// Sets the Z translation
// four units away
// from the view port.
this.matrix = (
                                                    [
1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, 0,
```

# 0, 0, -4, 1

]

);

// Return instance

// of GLEntity.

return this;

}

// GLEntity prototype

// declaration.

GLEntity.prototype = {

/**

* Get variables from the shaders

* and assign values to them.

* Prepare to download an Image

* file if the source path is

* not null.

*

* @param controller must include

```
    * a valid WebGLContext 'gl'
    * and a WebGLProgram 'program'.

    * @returns Number either zero
    * or one. One if an image file
    * must download.
    * Zero otherwise.
                                                */
getImageVariables: function(controller){

// Local WebGLContext.
var gl = controller.gl;

// Local WebGLProgram
var program = controller.program;

// Save the location
// of shader uniform
// named "u_sampler<n>"
// where n is the index
// of the texture
```

```
// represented by this GLEntity.
this.uSampler = gl.getUniformLocation
                                        (
    program,
    "u_sampler"+this.idx
                                        );

// Save the location
// of shader attribute
// named "a_tex_coord<n>"
// where n is the index
// of the texture
// represented by this GLEntity.
this.aTexCoord = gl.getAttribLocation
                                        (
    program,
    "a_tex_coord"+this.idx
                                        );

// If the shader includes
// an attribute for texels
```

```
// corresponding to
// this GLEntity,
// then call WebGL API
// vertexAttribPointer().
if (this.aTexCoord != null & &
    this.aTexCoord >= 0
    )
                                                {

// Parameters:
// 1. The attribute location
// within the shader
// to receive data from
// the bound WebGLBuffer.

// 2. Two array elements
// for each attribute.
// S and T texel coordinates.

// 3. Floating point type.
```

```
// 4. false:
// Don't normalize.
// Default buffer
// values expected
// in the range
// {0.0...1.0}

// 5. Stride in bytes
// between attributes.
// Default interleaved
// vertices with texels:
// X,Y,Z,S,T = 5 entries.
// Bytes per entry = 4.
// 5 * 4 = 20.

// 6. Offset from
// the start of the
// buffer
// X,Y,Z = 3 entries.
// Bytes per entry = 4.
// 3 * 4 = 12.
```

```
gl.vertexAttribPointer
                                        (
    this.aTexCoord,
    2,
    gl.FLOAT,
    gl.FALSE,
    20,
    12
                                        );

// Activate the
// attribute within
// the vertex shader.
gl.enableVertexAttribArray
                                        (
    this.aTexCoord
                                        );
                                        }

// If we have a source
// image file, then
```

```
// load it.
if (this.sSrc != null){

    // Create a new Image.
    this.img = new Image();

    // When the image's source
    // file loads, the 'onload'
    // event handler's 'this'
    // property equals the Image.
    // Create a 'controller'
    // property and assign
    // the GLControl reference.
    this.img.controller = controller;

    // Create an 'entity'
    // property and assign
    // this GLEntity.
    this.img.entity = this;

    // Activate setImage()
```

```
    // when the image file
    // downloads.
    this.img.onload = this.setImage;
    this.img.src = this.sSrc;

    // One more file
    // needs to download.
    return 1;
                                        }

// Zero files
// need to download.
return 0;
},
                                        /**
* Activates when
* the Image's source
* file downloads.
                                        *
* @param ev: Either
* Event Object
```

```
* or reference to GLControl.
* Value of 'ev'
* depends on who
* calls setImage().
                                                    */
setImage: function(ev){

// Assume the parameter
// directly references
// a controller.
var controller = ev;

var entity = null;

// If setImage() activated
// in response to an onload
// event, then ev.currentTarget
// represents the Image Object.
if (ev.currentTarget != null){

    // Save a reference
```

```
    // to the GLControl.
    controller = ev.currentTarget.controller;

    // Save a reference to
    // this GLEntity.
    entity = ev.currentTarget.entity;
                                                }

else {

    // For procedural
    // textures.
    // Save a reference to
    // the first GLEntity
    // from the list.
    entity = controller.aEntities[0];
                                                }

// Local WebGLContext.
var gl = controller.gl;
```

```
// Track the number
// of images loaded.
controller.nImagesLoaded++;

// In this case,
// Tell WebGL to
// process the texture
// unit TEXTURE0 + entity.idx,
// with the
// uniform sampler location
// entity.uSampler.
gl.uniform1i
                                        (
    entity.uSampler,
    entity.idx
                                        );

// Generate an empty
// WebGLTexture.
entity.texture = gl.createTexture();
```

```
// Must assign this
// texture as active
// before
// applying settings.
entity.setActiveTexture(entity,gl);

// Assigns the
// current texture
// as a 2D target.
// Only two options
// exist for targets:
// TEXTURE_2D and
// TEXTURE_CUBE_MAP.
gl.bindTexture
                                        (
    gl.TEXTURE_2D,
    entity.texture
                                        );

// Tells WebGL
// How to store
```

```
// pixels
// for use with
// texImage2D()
// and texSubImage2D().

// UNPACK_FLIP_Y_WEBGL
// required to
// map the top most
// part of an image
// to T=1.0.
gl.pixelStorei
                                        (
    gl.UNPACK_FLIP_Y_WEBGL,
    true
                                        );

// Exceptions thrown
// with texImage2D
// for browsers
// which don't support
// WebGL.
```

```
try{

// Procedural texture.
// No source file
// to down load.
// Instead use
// computer generated
// image data.
if (controller.nImagesToLoad == 0){

// texImage2D() includes two
// overloads. For procedural
// image data, use the
// following overload.
// Requires parameters
// for width and height.

// Parameter 1: target
// TEXTURE_2D for flat

// Parameter 2: 0
```

```
// the level of detail.

// Parameter 3: Number
// of color channels
// for this texture.

// Parameters 4 and 5
// specify width and height.
// The book's examples
// generate 32 x 32 square
// procedural image data.

// Parameter 6 represents
// the border.

// Parameter 7 represents
// the internal format
// and must match parameter 3.

// Parameter 8 represents
// the type of data provided.
```

```
// Procedural texture
// examples provided
// with the book
// use UNSIGNED_BYTE.

// Parameter 9
// contains the Uint8Array.
// When using Uint8Array
// Parameter 8 must match
// the data type.
// Uint8Array contains
// data of type UNSIGNED_BYTE.
gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGB,
    32,
    32,
    0,
    gl.RGB,
```

```
    gl.UNSIGNED_BYTE,
    entity.img
                                                    );
                                                    }

// An image file loaded
// for this texture.
else if(entity.img != null){

// For texture data
// from an Image Object
// use the following, simpler
// overload for texImage2D().

// Parameter 1: target
// TEXTURE_2D for flat
// graphical images.

// Parameter 2: 0 represents
// the level of detail.
```

```
// Parameters 3, and 4
// represent the the
// internal format and
// the pixel source data
// format. Both
// parameters must match.
// RGBA represents
// four channels
// including red, green, blue,
// and alpha.

gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    entity.img
                                                    );
                                                     }
                                                     }
```

```
catch(err){

// Exceptions thrown
// with cross origin
// attempts to
// use texImage2D().
controller.viewError
                                        (
    err,
    controller
                                        );

return;
                                        }

// Default wrap settings.
entity.setWrapToEdges(gl);

// Default minification
// and magnification filters.
```

```
entity.setMinMagFilters(gl);

// If the number of images
// to load have completed.
// Finish
// initialization with
// the controller.
if (controller.nImagesToLoad <= controller.nImagesLoaded){
controller.setProgramVariables
                                        (
   controller
                                        );
                                         }
},
                                        /**
Assign wrapping mode
for the texture.

Parameter CLAMP_TO_EDGE
stretches the texture
from the edge to edge
```

```
along the specified axis.

S represents the horizontal
axis of a texture.

T represents the vertical
axis of a texture.

@param gl: WebGLContext
                                        */
setWrapToEdges: function(gl){

// Clamp the horizontal texel
// to edges of defined polygons.
gl.texParameteri
                                        (
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE
                                        );
```

```
// Clamp the vertical texel
// to edges of defined polygons.
gl.texParameteri
                                                (
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE
                                                );

},
                                                /**
Assigns minification
and magnification filters
for the active texture.

NEAREST displays
the closest pixel.

@param gl: WebGLContext

                                                */
```

```
setMinMagFilters: function(gl){

// Minification filter.
// Fastest setting.
gl.texParameteri
                                        (
    gl.TEXTURE_2D,
    gl.TEXTURE_MIN_FILTER,
    gl.NEAREST
                                        );

// Magnification filter.
// Fastest setting.
gl.texParameteri
                                        (
    gl.TEXTURE_2D,
    gl.TEXTURE_MAG_FILTER,
    gl.NEAREST
                                        );
},
                                        /**
```

```
* Activate this
* GLEntity's texture.
* identified
* with a unit number.

* @param gl: WebGLContext
                                                */
setActiveTexture: function(entity,gl){

if (entity.img != null){

    // Khronos WebGL 1.0 specs
    // indicate enumerators starting
    // at TEXTURE0 increase
    // by one integer per
    // active texture.
    gl.activeTexture
                                                (
    gl.TEXTURE0 + entity.idx
                                                );
                                                }
```

```
}

}

/**

* Demonstrates mapping a cropped

* section of a graphic

* stretched across a square plane.

* Crop the section of the lighthouse

* image which displays the house itself

* close up.

* The upper left corner at (0.1,0.9)

* X coordinate 0 + 0.1 = 0.1

* Y coordinate 1 - 0.1 = 0.9

*

* The lower right corner is at (0.6,0.4)

* on the texture.

* 1.0 - 0.6 = 0.4 along the Y axis.

*

* This example loads

* either the lighthouse or

* the Butterfly fish graphic.

*
```

```
* Both textures display a
* cropped portion which is
* 50% the size of the
* original image.
*
* You may modify start and end positions
* to crop different portions,
* stretch or squash the
* original texture.
*
* @param s: String path
* to image file.

* @param b: Boolean
* 'true' means map the lighthouse.
* 'false' means map the fish.
*/
var GLSquareCrop = function(s,bL){

// Crop coordinates
var nXLeft, nXRight,nYTop,nYBottom = Number(0);
```

```
// The lighthouse
// photograph.
if(bL == true){

    // Left edge starts 10% from
    // the photograph's left edge.
    nXLeft = Number(0.1);

    // The right edge ends 60%
    // from the photograph's
    // left edge.
    // Covering 50% of the
    // width of the image.
    nXRight = Number(0.6);

    // The top edge stops
    // 90% from the photograph's
    // bottom edge.
    nYTop = Number(0.9);
```

```
    // The bottom edge starts
    // 40% from the photograph's
    // bottom edge.
    // Covering 50% of the
    // height of the
    // original photograph.
    nYBottom = Number(0.4);
                                                    }

// The fish graphic
// also crops out
// 50% of the original
// image.
// However the
// crop starts and
// ends along different
// edges.
else {
    nXLeft = Number(0.5);
    nXRight = Number(1.0);
```

```
    nYTop = Number(0.75);
    nYBottom = Number(0.25);
                                                    }


//Element array values.
var aIndices = new Uint16Array([
3,2,0,
```

**0,2,1,**

]);

// One square plane

// full size.

// The S texels

// are replaced with

// nXLeft and nXRight.

// The T texels

// are replaced with

// nYTop and nYBottom.

var aVertices = new Float32Array(

[

// left top front

// X,Y,Z:

-1.0, 1.0, 0.0,

// S,T:

nXLeft,nYTop,

// right top front

```
    // X,Y,Z:
    1.0, 1.0, 0.0,
    // S,T:
    nXRight, nYTop,

    // right bottom front
    // X,Y,Z;
    1.0, -1.0, 0.0,
    // S,T:
    nXRight, nYBottom,

    // left bottom front
    // X,Y,Z:
    -1.0, -1.0, 0.0,
    // S,T:
    nXLeft, nYBottom,
                                                ]
                                                );

// Entity with
// fish or lighthouse
```

```
// image for texture.
var aIm = new Array();
var n = new GLEntity(s,0);
aIm.push(n);

// New controller.
var data = new GLControl
                                                    (
    aVertices,
    aIndices,
    aIm,
    this
                                                    );
                                                     }
                                                    /**
* Demonstrates mapping a graphic
* full size across a square plane.

* @param s: String path
* to an image file.
                                                    */
```

```
var GLSquare = function(s){


// Typed array of
// unsigned integers.
// Index indirect reference
// to vertices and
// texels in 'aVertices'.
var aIndices = new Uint16Array([
// triangle 1
```

**3,2,0,**

// triangle 2

**0,2,1,**

]);

// One square plane

// Interleaved

// array of vertices

// The first three numbers

// represent one coordinate

// each for X,Y, and Z.

// The next two numbers

// represent S, T,

// texel units.

var aVertices = new Float32Array(

[

// left top

// index 0.

// X,Y,Z:

-1.0, 1.0, 0.0,

// S,T:

0.0,1.0,

// right top

// index 1.

// X,Y,Z:

1.0, 1.0, 0.0,

// S,T:

1.0, 1.0,

// right bottom

// index 2.

// X,Y,Z;

1.0, -1.0, 0.0,

// S,T:

1.0, 0.0,

// left bottom

// index 3.

// X,Y,Z:

-1.0, -1.0, 0.0,

// S,T:

0.0, 0.0,

```
]
);

// Create an
// array to contain
// GLEntity type.
var aIm = new Array();

// The first parameter
// to GLEntity is a string
// path pointing to an
// image file for use
// as a texture.
// The second parameter is
// an index used for
// the texture unit.
var n = new GLEntity(s,0);
aIm.push(n);

// Pass our vertices, indices
// array of GLEntity,
```

```
// and a copy of this 'class'
// to the
// controller.
// Every example in the
// book uses the same
// controller.
var controller = new GLControl
                                                    (
aVertices,
aIndices,
aIm,
this
                                                    );

                                                     }
                                                    /**
* Demonstrates mapping a graphic
* tiled four times
* across a square plane.
* Tiles twice along the X axis
* and twice along the Y axis.
```

```
                                                    *
* This example declares
* a center vertex, plus
* vertices to divide
* each edge of the square
* in half. This allows
* us to apply texels per
* vertex for tiling
* the graphic on
* each quadrant of the square.
* Such mappings aren't
* always necessary as seen
* in the GLSquareRepeat.js file.
                                                    *
* @param s: String path to
* an image file.
                                                    */
var GLSquareTile = function(s){

// Element index array
var aIndices = new Uint16Array([
```

```
// upper left quadrant
3,2,0,
```

# 0,2,1,

// upper right quadrant

7,6,4,

**4,6,5,**

// lower right quadrant

11,10,8,

**8,10,9,**

// lower left quadrant

15,14,12,

**12,14,13,**

]);

// Tiles display in four corners

// of the square plane.

// Interleaved vertices

// and texels require

// redundant vertex

// declarations.

var aVertices = new Float32Array(

[

// Top left quadrant:

// left top

-1.0, 1.0, 0.0,

0.0,1.0,

// right top

0.0, 1.0, 0.0,

1.0, 1.0,

// right bottom

0.0, 0.0, 0.0,

1.0, 0.0,

// left bottom

-1.0, 0.0, 0.0,

0.0, 0.0,

//Top right quadrant:

//left top

0.0, 1.0, 0.0,

0.0,1.0,

//right top

1.0, 1.0, 0.0,

1.0, 1.0,

//right bottom

1.0, 0.0, 0.0,

1.0, 0.0,

//left bottom

0.0, 0.0, 0.0,

0.0, 0.0,

//Bottom right quadrant:

//left top

0.0, 0.0, 0.0,

0.0,1.0,

//right top

1.0, 0.0, 0.0,

1.0, 1.0,

//right bottom

1.0, -1.0, 0.0,

1.0, 0.0,

//left bottom

0.0, -1.0, 0.0,

0.0, 0.0,

```
//Bottom left quadrant:
//left top
-1.0, 0.0, 0.0,
0.0,1.0,

//right top
0.0, 0.0, 0.0,
1.0, 1.0,

//right bottom
0.0, -1.0, 0.0,
1.0, 0.0,

//left bottom
-1.0, -1.0, 0.0,
0.0, 0.0,
                                                    ]
                                                    );

// Generic array.
var aIm = new Array();
```

```
// One entity with
// image file path.
var n = new GLEntity(s,0);
aIm.push(n);

// Controller
// constructor.
var data = new GLControl
                                                    (
aVertices,
aIndices,
aIm,
this
                                                    );
                                                     }
                                                    /**
* Demonstrates repeating a graphic
* across a square plane,
* with only four vertices.
* Uses REPEAT wrapping modes
```

```
* with WebGL API method
* texParameteri().

* @param s: String path
* to image file.
                                                */
var GLSquareRepeat = function(s){

// The same set of
// indices declared
// for a simple square
// plane.
var aIndices = new Uint16Array([
3,2,0,
```

**0,2,1,**

]);

// Identical X,Y,Z,

// coordinates as those

// used for the simple

// square plane.

// However S and T

// texels range from

// 0.0 to 4.0.

var aVertices = new Float32Array(

[

// left top

// X,Y,Z:

-1.0, 1.0, 0.0,

// S,T:

0.0,4.0,

// right top

```
// X,Y,Z:
1.0, 1.0, 0.0,
// S,T:
4.0, 4.0,

// right bottom
// X,Y,Z;
1.0, -1.0, 0.0,
// S,T:
4.0, 0.0,

// left bottom
// X,Y,Z:
-1.0, -1.0, 0.0,
// S,T:
0.0, 0.0,
                                                    ]
                                                    );

// Create a generic
// array to contain
```

```
// GLEntity type.
var aIm = new Array();

// The first parameter
// to GLEntity is a string
// path pointing to an
// image file for use
// as a texture.
// The second parameter uses
// to identify the
// texture unit.
var n = new GLEntity(s,0);

aIm.push(n);

// Instantiate GLControl
// the same as the book's
// other examples.
var data = new GLControl
                                                    (
aVertices,
```

```
aIndices,
aIm,
this
                                        );
                                        }

                                        /*
* Prototype 'class'
* for repeating
* images on a texture.
                                        */
GLSquareRepeat.prototype = {
                                        /**
Assign wrapping modes
for the texture.
REPEAT which
displays tiles.
along the specified axis.

S represents the horizontal
axis of a texture.
```

```
T represents the vertical
axis of a texture.

init() activates after
textures download activate
and initialize.

@param: controller GLControl
'class'.
                                        */
init:function(controller){
var gl = controller.gl;
var entity = controller.aEntities[0];

//Repeat the horizontal texel
//along edges of
//defined polygons.
gl.texParameteri
                                        (
gl.TEXTURE_2D,
```

```
gl.TEXTURE_WRAP_S,
gl.REPEAT
                                        );

//Repeat the vertical texel
//along edge of
//defined polygons.
gl.texParameteri
                                        (
gl.TEXTURE_2D,
gl.TEXTURE_WRAP_T,
gl.REPEAT
                                        );
},
                                        }
                                        /**
* Procedural graphic.
* Procedural graphics are
* computer generated.
* Use the drop down menu
* to select specific methods
```

```
* to create different
* graphics for use as textures.
                                                    */
var GLTexProcedure = function(){

// The same indices
// as declared for
// the simple square plane.
var aIndices = new Uint16Array([
3,2,0,
```

**0,2,1,**

]);

// The same vertices

// and texels as

// declared for the

// simple square plane.

var aVertices = new Float32Array(

[

// left top

-1.0, 1.0, 0.0,

0.0,1.0,

// right top

1.0, 1.0, 0.0,

1.0, 1.0,

// right bottom

1.0, -1.0, 0.0,

1.0, 0.0,

```
// left bottom
-1.0, -1.0, 0.0,
0.0, 0.0,
                                                    ]
                                                    );

// Every array contains
// 3 color channels for a square
// 32 pixels wide by 32 pixels high.
// 32 *32* 3 = 3072.
// Use the same size buffer
// for each generated texture.
this.nBuffer = Number(3072);

// We'll display
// to the Web page,
// the
// first 32 entries
// for each texture.
// Save a reference to
```

```
// the HTML element
// to display text.
this.eDebug = document.getElementById
                                        (
"eDebug"
                                        );

var aIm = new Array();

// The image file path
// String is null.
// We don't need to
// load an Image file.
var n = new GLEntity(null,0);

// Move the square
// mesh higher on
// the canvas.
n.matrix[13] = 0.5;

// We'll start with the
```

```
// blue green gradient.
// Assign a Uint8Array
// filled with BYTE data
// representing the gradient.
n.img = this.generateGradientBG();
aIm.push(n);

// Instantiate GLControl
// with the same parameters
// as the other examples.
var controller = new GLControl
                                        (
aVertices,
aIndices,
aIm,
this
                                        );

// Stop processing
// if there was an error.
if (controller == null){
```

```
    // GLControl
    // should have
    // displayed an
    // error message
    // for the user.
    return;
                                        }

// Assign a listener
// to the drop down
// select menu.
this.setListener(controller);

                                        }

                                        /**
* Prototype for generating
* some procedural textures.
                                         */
GLTexProcedure.prototype = {
                                        /**
```

```
* Assign a listener
* to the select option
* drop down menu.
* @param controller: GLControl reference.
                                                    */
setListener: function(controller){

// Obtain a reference to
// the Web page's select element.
var menu = document.getElementById("mSelect");

// Add this example's method
// named optionSelect()
// to the 'click' event
// listener for the menu.
menu.addEventListener
                                                    (
    'change',
    this.optionSelect,
    false
                                                    );
```

```
// The currentTarget
// of the event object
// will maintain a reference
// to the GLControl object.
menu.controller = controller;
},
                                        /**
* Respond to selection from
* the option drop down menu.
* Calls methods to generate
* and display color
* data by procedure.
                                        *
* We only need new
* Uint8Array data,
* assigned
* to the img property
* of the first GLEntity.
                                        */
optionSelect: function(ev){
```

```
// The currentTarget is the
// select menu.
var controller = ev.currentTarget.controller;

// The controller always
// saves a reference
// to the current example
// project's "class" prototype.
var glDemo = controller.glDemo;

// We only modify the first
// and only GLEntity in the
// array.
var entity = controller.aEntities[0];

// Retrieve the
// selected menu option.
var s = ev.currentTarget.value;

// For the option selected
// change the first GLEntity's
```

```
// image data.
// Each generate<Type> returns
// a Uint8Array filled with
// data representing red, green,
// and blue channels for
// a 32 x 32 square texture.
switch(s){
case "Blue Green Gradient":
    entity.img = glDemo.generateGradientBG();
break;

case "Four Tiles":
    entity.img = glDemo.generateTilesRGB();
break;

case "Red Blue Stripes":
    entity.img = glDemo.generateStripesRB();
break;

case "Red Square":
    entity.img = glDemo.generateSquareR();
```

```
break;

case "Red Green Random Colors":
    entity.img = glDemo.generateRandomRG();
break;

                                                    }

// No rotation on the
// procedural texture
// for demonstration purposes.
controller.nRad = Number(0);

// Call setImage() directly.
// Other examples trigger
// setImage() with an
// onload event.
entity.setImage(controller);

// Set location
// to better view
```

```
// canvas.
// iPhone neglects
// to reset zoom.
window.location.assign('#mSelect');
},
                                                    /**
* Create values to display
* a solid red color.
* Uint8Array represents an
* array of integers.
* Each integer stays within
* range {0,255}.
* Each integer represents one
* color channel.
                                                     *
* @return Uint8Array representing
* color data by channels.
                                                    */
generateSquareR: function(){

// New Uint8Array
```

```
// of size nBuffer.

var u8a = new Uint8Array(this.nBuffer);

// Loop over every three
// values.
for (var i = 0; i < this.nBuffer; i+=3){

    // The first channel
    // represents red
    // at the maximum value:255.
    u8a[i] = 255;

    // Default values
    // equal zero.
                                        }

// See the first 32 values in
// the new texture.
this.printArray
                                        (
```

```
    "Red Square",
    u8a
                                                );

// Return the Uint8Array
// for use as WebGL texture data.
return u8a;
},
                                                /**
* Create vertical stripes,
* alternating red and blue colors.
                                                 *
* @return Uint8Array representing
* color data by channels.
                                                 *
* Every three Bytes alternate
* colors.
                                                */
generateStripesRB: function(){

var u8a = new Uint8Array(this.nBuffer);
```

```
// The for loop iterates over
// every three values.
for (var i = 0; i < this.nBuffer; i = i+3){
    var j = i % 2;

    // Red stripe:
    if (j == 0){
    u8a[i] = 255; // Red
    u8a[i+1] = 0;
    u8a[i+2] = 0;
                                                    }

    // Blue stripe:
    else {
    u8a[i] = 0;
    u8a[i+1] = 0;
    u8a[i+2] = 255; // Blue.
                                                    }

                                                    }
```

```
this.printArray
                                                    (
    "Red Blue Stripes",
    u8a
                                                    );
return u8a;
},
                                                    /**
* Create texture data
* with four different
* colored quadrants.
* The tiles are solid red,
* green, blue, and violet.
                                                    *
* @return Uint8Array representing
* color data by channels.
                                                    *
* Find left or right side:
* Each row is 32 pixels wide
* with three color channels
```

*32* 3 = 96.

* Left side < 48.

* Right side >= 48.

*

* The full size of the array is

*32* 32 * 3 = 3072.

* Top half < 3072.

* Bottom half >= 3072.

*

*/

generateTilesRGB: function(){

// Fills default alpha value

var u8a = new Uint8Array(this.nBuffer);

// Quadrant booleans.

var bTop = true;

var bLeft = true;

var j = Number(0);

var nHalf = this.nBuffer/2;

```
for (var i = 0; i < this.nBuffer; i = i+3){

// Default black:
nRed = Number(0);
nGreen = Number(0);
nBlue = Number(0);

// pixels display
// in the top half if
// less than
// half the size
// of the entire buffer.
bTop = i < nHalf;

// j = each column.
var j = i % 96;
bLeft = j < 48;

// Top tiles.
if (bTop){
```

```
// Top left tile:
if (bLeft){
    nRed = Number(255);
                                                }
// Top right tile.
else {
    nBlue = Number(255);
                                                }
                                                }

// Bottom tiles.
else {

// Bottom left.
// Blue+Red=Violet.
if (bLeft){
    nRed = Number(255);
    nBlue = Number(255);
                                                }

// Bottom right tile.
```

```
else {
    nGreen = Number(255);
                                                }
                                                }

// Assign three
// color channels
// for one pixel.
u8a[i] = nRed;
u8a[i+1] = nGreen;
u8a[i+2] = nBlue;
                                                }

// Display first
// 32 values.
this.printArray
                                                (
    "Four Tiles",
    u8a
                                                );
return u8a;
```

},

/**

* Generates light blue green

* gradient from left to right.

* Just change values for channels

* to modify gradient colors.

*

* @return Uint8Array representing

* color data by channels.

* One row is represented

* by 32 pixels with 3 color

* channels each.

*32* 3 = 96.

j ranges {0..95}

k ranges {95..0}

blue's range: {255..160}

green's range: {160..255}

red is always zero.

As blue decreases

```
    green increases, for each row.
                                                */
generateGradientBG: function(){
    // Fills default alpha value
    var u8a = new Uint8Array(this.nBuffer);

    var j = Number(0);
    var k = Number(0);

    for (var i = 0; i < this.nBuffer; i = i+3){

    // Each row r,g,b{0..95}
    // Each row one pixel {0..31}
    j = i % 96;
    k = 95 - j;

    // Assign Red, Green,
    // Blue values:
    u8a[i] = 0;
    u8a[i + 1] = 255-k;
    u8a[i + 2] = 255-j;
```

```
                                        }

    this.printArray
                                        (
    "Blue Green Gradient",
    u8a
                                        );

    return u8a;
},
                                        /**
* Create texture data
* for random red and green pixels.
                                         *
* @return Uint8Array representing
* color data by channels.

* Math.random() returns values
* between 0 and 1.
* Multiply by 255 for
```

```
* ranges between 0 and 255.
                                                */
generateRandomRG: function(){

    var u8a = new Uint8Array(this.nBuffer);

    for (var i = 0; i < this.nBuffer; i = i+3){

    // Red
    u8a[i] = Math.random()*255;

    // Green
    u8a[i + 1] = Math.random()*255;

    // Blue
    u8a[i + 2] = 0;
                                                }

    this.printArray
                                                (
    "Red Green Random Colors",
```

```
    u8a
                                                    );


    return u8a;
},
                                                    /**
* Display the first 32 values
* in Uint8Array.
* Red, green, then blue.
* No alpha channel supplied for
* these textures.

* @param s: String.
* @param a: Uint8Array.
                                                    */
printArray: function(s,m){
    this.eDebug.innerHTML +="First 32 Values For: "+s+"<br />";
    for (var i = 0; i < 33; i++){
    if (i % 3 == 0)
    this.eDebug.innerHTML += "<br />RBG: ";
    this.eDebug.innerHTML += a[i]+" | ";
```

```
}

}

}
```

# Source Code Downloads

You may download all the source code and graphics for the book's examples. The directory is password protected. Follow the link to v.php, then enter user name "ebook" and password "code". The download page includes a number of options including the ability to download everything in one zipped file "v.zip", view fully commented JavaScript, view the example projects. The zip file includes JavaScript files without comments as well.

# Copyright

Title:"WebGL Textures & Vertices: Beginner's Guide"

Copyright: A. Butler

Published: 6/22/2015

Publisher: Seven Thunder Software

Copyright (C) 2015 A. Butler. All rights reserved.

No part of this publication except the JavaScript, HTML, and CSS software code (the "Source"), may be reproduced, stored in a retrieval system, copied in any form or by any means, electronic, mechanical, photocopying, recording or otherwise transmitted without written permission from the publisher. You must not circulate this book in any format.

Permission is hereby granted, free of charge, to any person purchasing or borrowing a copy of "WebGL Textures & Vertices: Beginner's Guide" from Amazon.com, Inc., to use the Source, including without limitation the rights to use, copy, modify, merge, publish, or distribute, and to permit persons to whom the Source is furnished to do so, subject to the following conditions:

The Source may not be used to create training material of any sort, including courses, books, instructional videos, presentations, *etc.* without the express written consent of A. Butler.

THE SOURCE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This book is licensed for your personal enjoyment only. This e-book may not be resold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. Thank you for respecting the hard work of this author.

Enjoy free tutorials and learn about upcoming books online at www.SevenThunderSoftware.com.